Onboard Artificial Intelligence for Space Situational Awareness with Low-Power GPUs

Michael Lim, Payam Mousavi, Jelena Sirovljevic, Huiwen You
MDA Systems

ABSTRACT

Onboard processing provides the reduction in latency that is critical to Space Situational Awareness (SSA) applications. By receiving data directly from the sensor and processing it in real-time on board of spacecraft, this technology enables real-time processing and response. Currently, onboard processing is a nascent technology and the capabilities that exist are limited: they are highly-customized, one-off systems typically built for large spacecrafts. This is poised to change drastically in the coming decade, as off-the-shelf computing hardware and algorithms mature while spacecraft operations turn towards more scalable small-sat missions. Within that time period, onboard processing will transform from a research topic to an essential element of most space missions.

One of the key functionalities for onboard processing in SSA domain is object classification. Current state of the art classification algorithms are based on Artificial Intelligence (AI) technologies. Due to prohibitively high computational needs for AI applications, their deployment onboard a spacecraft has not been possible to date. However, the rapid advance in AI-oriented computing hardware, especially Graphics Processing Units (GPU), has opened the door to AI in space. In particular, low size, weight, and power (SWaP) GPU devices have been developed that would be ideal for space-based processing.

MDA is currently investigating use of state of the art off-the-shelf low-power GPUs for deployment of AI applications essential for real-time object identification as part of the SSA domain. This work is motivated by recent increased focus within the AI community on operationalizing AI methodology. In this paper, we discuss the motivation behind the research, technical details of the implementation and current results. More specifically, three simple neural networks of different sizes trained on MDA Sapphire dataset for space object classification are presented. The classification performance of these algorithms are benchmarked demonstrating that larger models tend to be more robust to added noise in the input image. Effects of computing optimization techniques applied on these models are also presented, which generally show great improvement by several factors in throughput and power efficiency of these algorithms onboard the hardware platform hosting these technologies.

1. INTRODUCTION

In-orbit threats to satellites are rapidly increasing and evolving. With close to two thousand satellites currently in orbit in addition to several hundreds of thousands of debris objects, the risk of unintentional collisions between space objects is heightened. These numbers are expected to increase threefold in the next decade. Another new emerging threat is hostile behavior in space from foreign nations or rogue organizations. Therefore, it is critical to take measures to safeguard the assets against in-orbit threats.

Traditionally, Resident Space Object (RSO) tracking and surveillance employ high latency sense-evaluate methods optimized to extract as much information as possible from each data sample. These methods rely on preset parameters and *a priori* calculations to control system behavior. A typical concept of operations for traditional Space Situational Awareness (SSA) capabilities include planning the observations days in advance, and downlinking data to the ground for processing on server environments. For routine observations, this process may be automated. However, for situations that include unanticipated changes in RSO behavior or a close approach of two RSOs, time-consuming evaluation and analysis by human operators is required.

Consequently, the current SSA tracking and surveillance methods are insufficient for the newly emerging space environment. The projected increase in the number of RSOs over the next decade also increases the risk of collisions and number of close approaches between RSOs. It will be infeasible to track and resolve all the collision warnings using existing manual methods. To protect space assets, response times to detect and warn spacecraft operators of imminent threats will need to decrease by an order of magnitude.

Onboard processing refers to the task of deploying an algorithm or application on the platform where payload data is generated or acquired, and is a critical component of reduction of latency and providing responses in real-time. This technology is therefore a key capability required for SSA applications that produce timely results, as is for myriad of other space and defense applications, where platforms, such as satellites, are often deployed into a heavily constrained operational environments, and where transmitting the data to an off-site location for further processing poses a critical bottleneck in conducting the mission. In order to deploy sophisticated SSA applications in Size, Weight and Power (SWaP)-constrained environments, such as spacecraft platforms, solutions are needed that satisfy both the real-time performance and SWaP requirements. With recently increased interest in Artificial Intelligence (AI) algorithms and their deployment in commercial applications such as autonomous vehicles or Internet-of-Things (IoT)/mobile computing, low-power GPUs have been developed that optimize performance while being mindful of the SWaP limitations.

As for Space Surveillance algorithms, European Space Agency (ESA) [1], National Aeronautics and Space Administration (NASA) [2] and, more recently, Australian Space Agency [3] have all recognized the potential of Machine Learning (ML) and AI for space applications. Deployment of these applications into space has also been an active area of research in the last few years. There are many aspects to consider when deploying these kinds of applications into space, including security and data management [4], environmental effects on components [5], [6] and consequences of operating in SWaP-constrained environments. The focus of the research presented here is to address the latter.

More precisely, we investigate the utilization of the low-power GPU technology for deployment of AI-based applications for efficient classification of RSOs and other objects onboard of a surveillance-of-space spacecraft. We present Deep Neural Network (DNN) models that perform object classification of space objects in the presence of different quantities of noise added to the input images. We then examine various types of computing optimizations and apply those techniques on the models discussed to present performance benchmarks before and after these optimizations are applied. We present our benchmarks in terms of throughput speed and computing efficiency of our baseline and optimized models.

2. SPACE OBJECT CLASSIFICATION

Onboard space object classification is an essential precursor to a range of future potential capabilities, such as onboard image pre-processing, anomaly detection, intent identification, as well as autonomous target tracking and Course-of-Action (COA) generation. We therefore begin by exploring the state of the art supervised classifiers [7] that are based on Deep Neural Networks (DNN). Our focus is not only to develop and optimize a classifier, but to investigate what is required to deploy models onboard of low-power GPUs. To assess our classification performance and to further motivate the need for larger models deployed onboard, we inject different levels of white Gaussian noise to the images prior to the training and classification, expectedly degrading the model performance. We then demonstrate that larger models can achieve much better performance (compared to smaller models) in the presence of substantial noise that is likely to be present in images collected by smaller satellites.

2.1 Dataset

To develop Machine Learning algorithms for the simple task of space object classification, we leverage an internally curated dataset collected in a single day by MDA-operated Sapphire [8], a Canadian space surveillance satellite that was launched in 2013. Sapphire is a military satellite designed to monitor space debris as well as satellites within an orbit of approximately 6,000-40,000 kilometers. An example of imagery collected by Sapphire is shown in Fig. 1. The geostationary satellites (i.e., RSOs) appear circular, while the stars (that are moving with respect to the satellite) appear as streaks as the image frames are integrated in time.

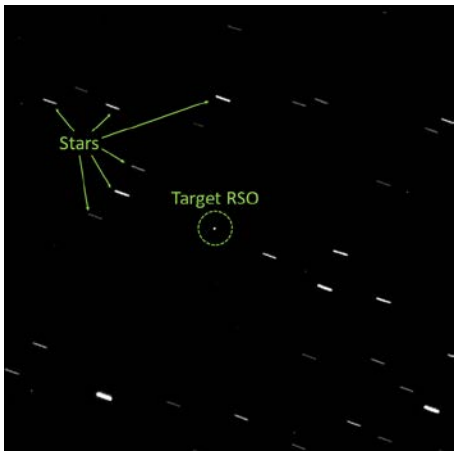


Fig. 1 – An example Sapphire Imagery. RSOs in geostationary orbit appear circular, while stars appear as streaks as the images are integrated in time

There are many challenges associated with the pre-processing, detection and classification of objects of interest in the raw imagery. Some of these challenges are demonstrated in Fig. 2. For comparison, a clean image is shown in Fig. 2a, while Fig. 2b shows a noisier image captured over the South Atlantic. Sensitive cooled Charge-Coupled Device (CCD) detectors pick up radiation strikes during image acquisition. Fig. 2c shows an example with substantial stray light from out-of-field bright sources (e.g., Earth, Moon, and Sun). In Fig. 2d, a post-processed image is shown, where all green pixels have passed through the filtering process while the black and grey pixels are removed. This is accomplished using detection and classification techniques described in this work.

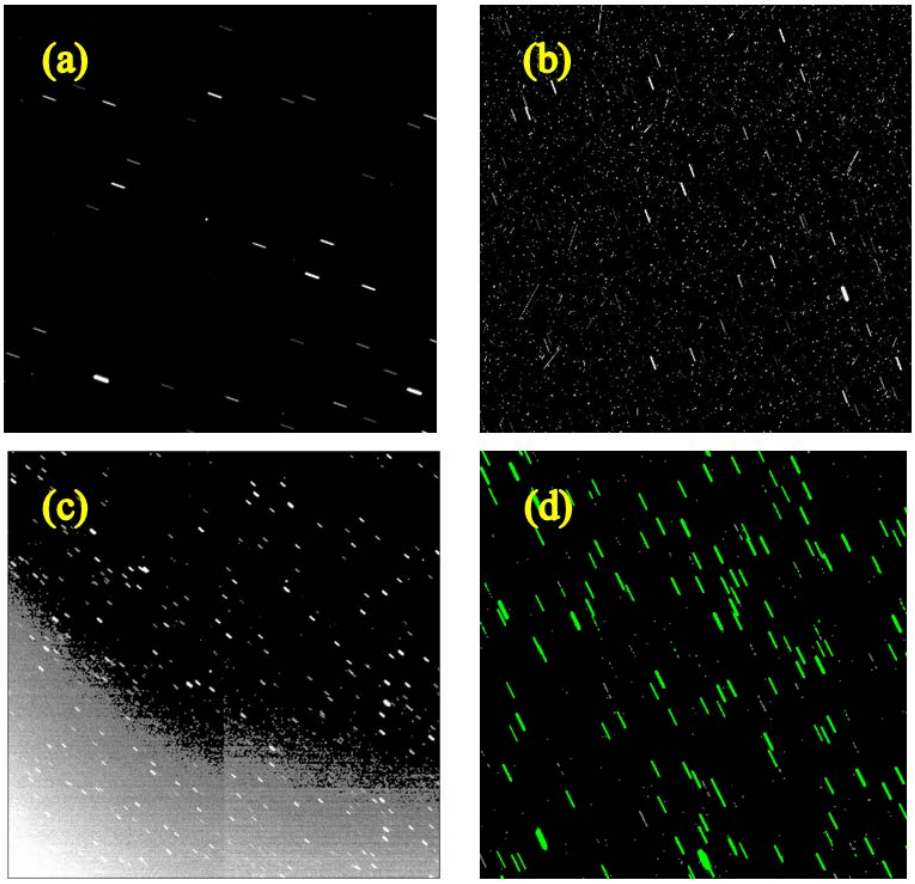


Fig. 2 - (a) A clean Sapphire image, (b) Image collected over South Atlantic: Radiation strikes, (c) Stray light contamination, (d) Filtered image: Green pixels pass through while grey and black are filtered out.

The first step is to create a labeled dataset ready for ingestion by an ML model (i.e., a DNN in this case). Five classes of objects were selected for this study. These are:

1. Resident Space Objects (RSO)
2. Stars
3. Stray Light
4. Dark current defect (DEF) – i.e., hot pixels
5. Transient Radiation defects (RAD)

A database of raw Sapphire imagery collected over the course of a day is mined to automatically crop out (and label) 9×9 images containing the 5 classes of objects of interest. These images were manually cleaned to ensure labelling accuracy. To detect objects, a sliding window approach is used. Historical Two-Line Element (TLE) data is used to verify the location of RSOs in the images. An iterative process is initiated by first manually labeling a small set of images. Simple classifiers are then trained on this small dataset. This process is repeated multiple times, followed by manual cleaning, to create the final large dataset. Randomly selected samples are shown in Fig. 3.

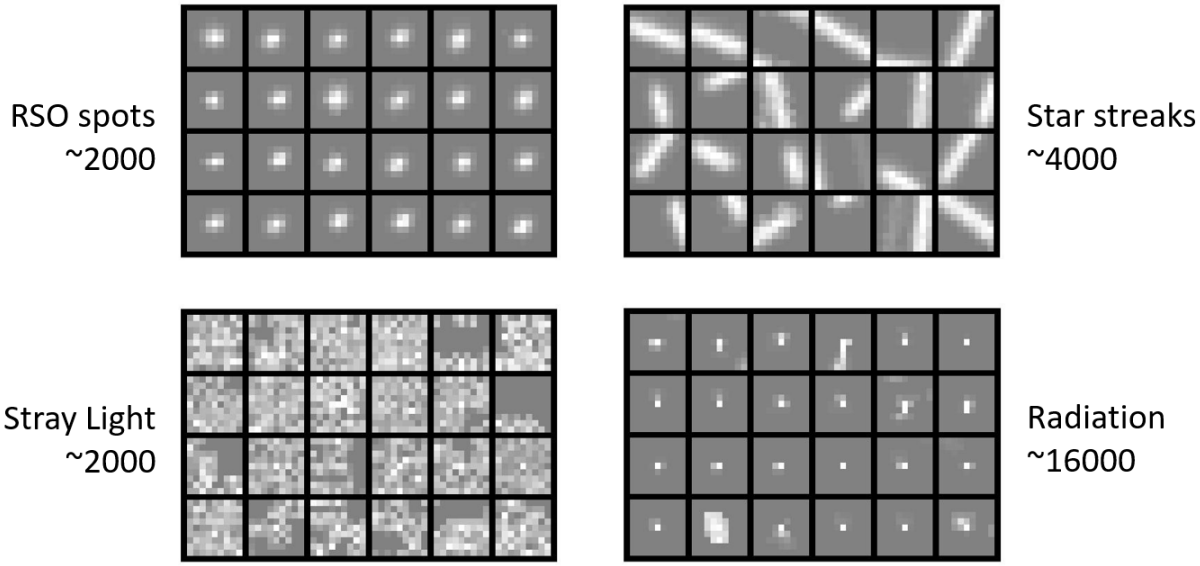


Fig. 3 - Randomly selected labeled images

Each image crop is further normalized by the maximum pixel intensity for that image. This is done to ensure that the pixel intensity values are approximately at the same range for all images, allowing for more efficient optimization of our models. Note that this will result in some loss in intensity information as the differences between intensities of different object classes cannot be used to improve classification accuracy. However, we observe that geometrical differences in the classes are sufficient to distinguish them accurately. In the next section, we provide more details on the model architectures as well as discuss the chosen metrics for evaluating performance.

2.2 Models and Metrics

For classification of 'clean' images (i.e., no added noise), we compare the performance of two simple feedforward neural networks: a Fully Connected Network (FCN-1) (see Table 1), and a Convolutional Neural Network (CNN-1) (see Table 2).

Table 1- FCN-1: Fully Connected (Dense) Network Architecture

Input: 81×1 (Flattened 9×9 Intensity image)
Dense Layer(INPUT-DIM=81, OUTPUT-DIM=32) + ReLU Activation
Dropout (p = 0.5)
Dense Layer(INPUT-DIM=32, OUTPUT-DIM=5) + Softmax Activation
Total Number of Parameters: 2,789

Table 2 – CNN-1: Convolutional Neural Network Architecture

Input: 9×9×1 (Intensity images)
2D Convolution Layer(OUT-CHANNEL=16, KERNEL=3, STRIDE=1, PADDING=1) + ReLU Activation
Dropout (p = 0.5), *Flatten*
Dense Layer(INPUT-DIMENSION=9*9*16, OUTPUT-DIMENSION:=5) + Softmax Activation
Total Number of Parameters: 6,645

Traditionally, CNNs [9] are expected to perform better at image classification as they encode the translation-invariance of objects within an image. Fully-connected (i.e., Dense) networks, while simpler, are more sensitive to shifts in the position of the object of interest, and could produce poor results if the object is expected to be at different locations.

A third larger CNN model (CNN-2) described in Table 3 is used to investigate whether or not using larger networks results in improved classification performance in the presence of noise.

Table 3 – A larger CNN Architecture

Input: 9×9×1 (Intensity images)
2D Convolution Layer(OUT-CHANNEL=16, KERNEL=3, STRIDE=1, PADDING=1) + ReLU Activation
2D Convolution Layer(OUT-CHANNEL=32, KERNEL=3, STRIDE=1, PADDING=1) + ReLU Activation
Dropout (p = 0.5), *Flatten*
Dense Layer(INPUT-DIMENSION=9*9*32, OUTPUT-DIMENSION=5) + Softmax Activation
Total Number of Parameters: 17,765

For our experiments, to train FCN-1 and CNN-1, we use the Adam optimizer [10] with a learning rate of 0.04 for 200 epochs. To train the larger CNN-2 model, we use the SGD Optimizer [11] with a learning rate of 0.006, for 500 epochs. The images are randomly shuffled and 25% are saved for testing (i.e., not used during training). All metrics reported will be on the test set. Here our aim was to utilize simple models that produce acceptable results. As the purpose of this work is to experiment with onboard deployment of neural networks, we will leave the optimization of the model architecture and hyperparameters to future work.

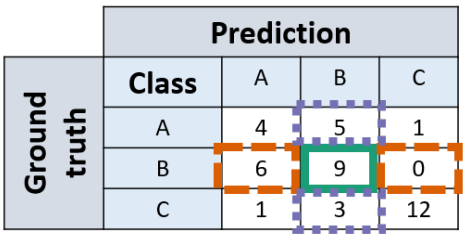
To evaluate performance, in addition to reporting the overall classification accuracy we use the average *F1-Score* [12] as defined in Fig. 4:

$$Precision = \frac{TP}{TP + FP} \quad , \quad Recall = \frac{TP}{TP + FN}$$

$$F1 = 2.\frac{Precision.Recall}{Precision + Recall}$$

Fig. 4 – Definition of Metrics: TP – True Positive, FP – False Positive, FN – False Negative. The F1-Score is the harmonic mean of Precision and Recall

The *F1-Score* is a more appropriate metric compared to the overall accuracy as it accounts for the potential class imbalances. Intuitively, the *F1-Score* captures a combination of precision and recall and its value ranges from 0 to 1 (i.e., perfect precision and recall). Fig. 5 illustrates how the precision and recall are calculated for an example classification task.



$$\boldsymbol{precision} = \frac{\boldsymbol{TP}}{\boldsymbol{TP} + \boldsymbol{FP}} \qquad \boldsymbol{recall} = \frac{\boldsymbol{TP}}{\boldsymbol{TP} + \boldsymbol{FN}} \qquad \boldsymbol{F1} = \frac{\boldsymbol{2} \times \boldsymbol{TP}}{\boldsymbol{2} \times \boldsymbol{TP} + \boldsymbol{FP} + \boldsymbol{FN}}$$

Fig. 5 - Illustration of how precision and recall are extracted from the rows and columns of the confusion matrix and how the F1-score is calculated.

Details of the experiments and the results are reported in the following section.

2.3 Experiments and Results

Before onboard deployment, the models are trained and tested under different conditions. The experiments are designed to provide insight into the training data distribution as well as help compare the performance of proposed models with different quantities of added Gaussian noise. This is done to assess the robustness of the models in case of lower signal-to-noise ratio (SNR) images (expected from smaller satellites).

Following the training procedures described before, models FCN-1 and CNN-1 are trained on the original dataset (i.e., no noise added) as well as the dataset with additional Gaussian noise (with $\mu = 0$, $\sigma = [0.05, 0.10, 0.20]$). After adding noise, the pixel intensities are clipped to ensure they are positive and normalized to one. As an illustration, Fig. 6 shows example images from each class with different levels of simulated noise added. We observe that DEF and RAD classes become increasingly indistinguishable as noise levels increase.

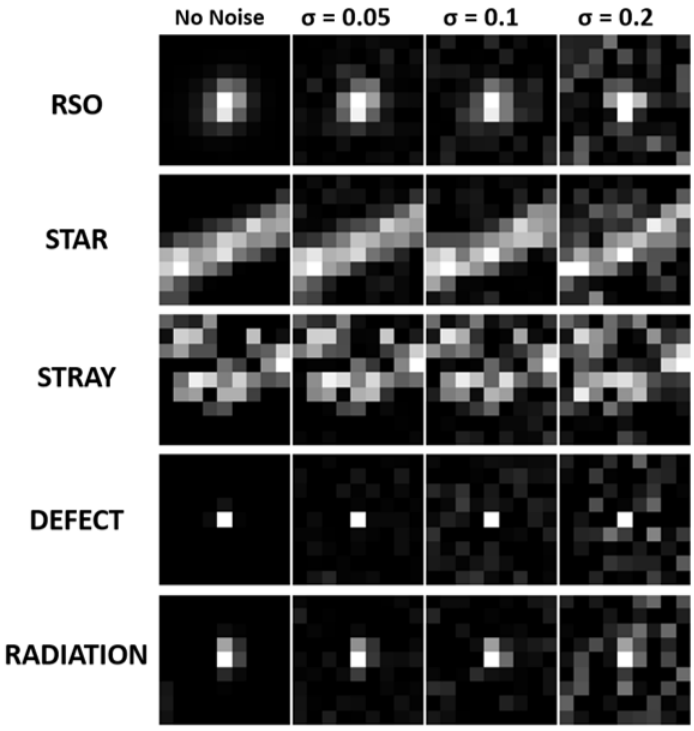


Fig. 6 - Different quantities of Gaussian noise (with μ=0) added to one sample from each class

The resulting confusion matrices (as evaluated on the test set) are shown in Fig. 7.

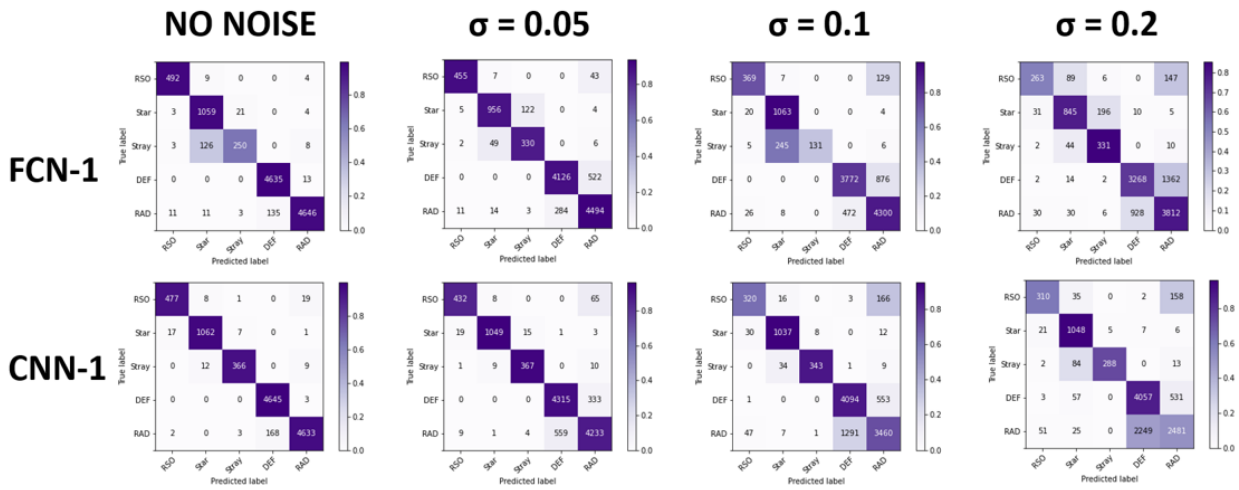


Fig. 7 – Test set Confusion Matrices for the models FCN-1 and CNN-1 trained on the same dataset but with different levels of noise

As expected, both models perform best on 'clean' images, and the performance is degraded with increasing levels of noise. Moreover, we note that DEF and RAD classes are more often confused as the noise levels increase, while STARS are still reliably classified. This is due to the STARS having distinct geometric signatures (i.e., angled lines) compared to RAD and DEF which start looking very similar as the noise levels increase.

The Average *F1-Scores* for the 5 classes are shown in Fig. 8,

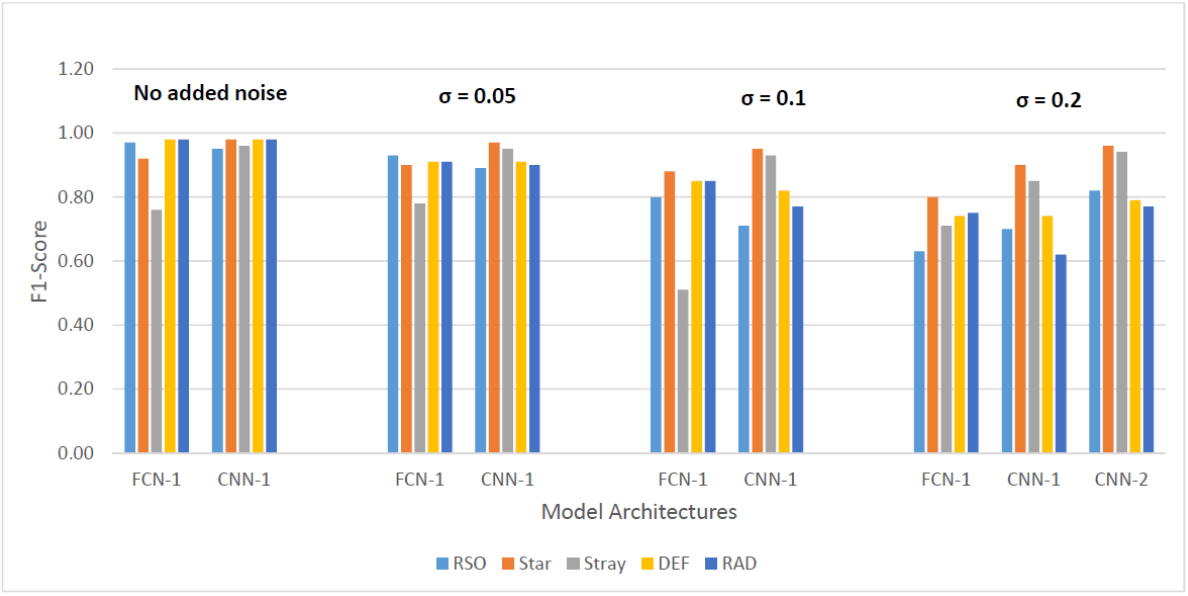


Fig. 8 - Comparing F1-Scores of each class for the three models. Note the significant performance improvement in case of the larger model CNN-2 in the presence of significant amount of noise

As expected, with increasing noise levels, the *F1-Scores* decrease. Overall, the CNN-1 model performs slightly better, especially as noise levels increase. The larger model (CNN-2) performs significantly better (especially for STARS and RSOs) in the case with the largest quantity of added noise. The same results are shown in Table 4.

Table 4 - Summary of F1-Score results for the models trained with increasing levels of added Gaussian noise

	No Noise		σ=0.05		σ=0.1		σ=0.2		
Classes	**FCN-1**	**CNN-1**	**FCN-1**	**CNN-1**	**FCN-1**	**CNN-1**	**FCN-1**	**CNN-1**	**CNN-2**
RSO	0.97	0.95	0.93	0.89	0.80	0.71	0.63	0.70	0.82
Star	0.92	0.98	0.90	0.97	0.88	0.95	0.80	0.90	0.96
Stray	0.76	0.96	0.78	0.95	0.51	0.93	0.71	0.85	0.94
DEF	0.98	0.98	0.91	0.91	0.85	0.82	0.74	0.74	0.79
RAD	0.98	0.98	0.91	0.90	0.85	0.77	0.75	0.62	0.77

We conclude that, while for our curated Sapphire dataset, simple CNN models suffice to produce good classificaiton results, as the noise levels increase (as anticipated in case of smaller satellites), larger CNN models could be used to mitigate lower SNR. As the size of the models increase, optimization prior to onboard deployment becomes more important. The details of these optimizations are discussed next.

3. OPTIMIZATION METHODS FOR ONBOARD PROCESSING

There are many operational challenges to deploying computationally intensive algorithms onboard platforms, ranging from external influences such as mechanical, thermal or radiation effects to reduced computing power compared to other means of processing the data. Particularly for DNN algorithms that typically require large computing bandwidth, deploying the algorithms onboard likely requires large power to support dedicated hardware for computing, such as a GPU, which in many vehicles (e.g., solar-powered satellites) is simply not available or is available at a premium.

Because of this compute-intensive nature of neural network models, it is often desirable to optimize the computing labour the hardware carries out, especially in an operational context. This can further improve the efficiency of the algorithm and ultimately ease the computing resource requirement for the system that can be difficult to fulfill in

space operational context. Optimization can be carried out at different levels of the application stack as well as the hardware the application runs on.

Machine learning frameworks often utilize compute Application Programming Interfaces (API) and hardware drivers to utilize external hardware for performing lower-level computations of a neural network algorithm. This is to allow for portability of those frameworks across different hardware. Deep learning frameworks such as TensorFlow or PyTorch implement layer-specific operations using a compute API defined protocol. Compute APIs, such as Nvidia CUDA or OpenCL, translate these operations to hardware-specific compute kernels that a target hardware can carry out. Lastly, the hardware itself may be specifically designed to carry out certain instructions quickly or efficiently.

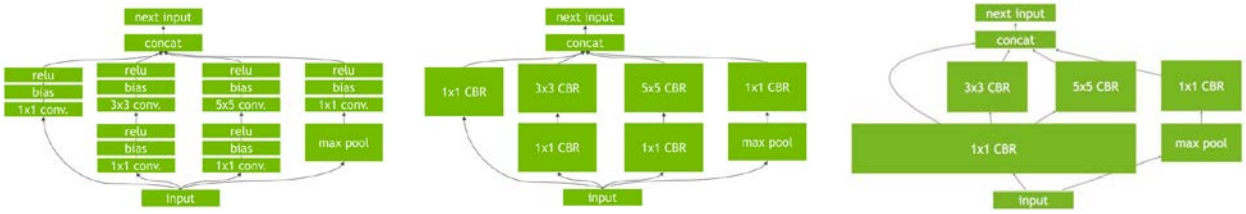
Optimization can be carried out at any level of translation. However, this comes at the cost of portability, largely because optimization at a particular level is in reference to some specific implementation at a lower level, and therefore may not be exploited by a different implementation, as will be discussed in this section.

3.1 Compute Kernel Optimization

Compute kernels are small execution units the hardware carries out to perform a certain mathematical function, such as scalar and matrix operations, element-wise operations, etc. Each layer in a neural network is comprised of many compute kernels. In practice, kernels are implementations of very basic and most commonly used mathematical operators. This allows for many different scientific applications to utilize these kernels to implement respective algorithms.

However, utilizing a combination of such generic implementations adds overhead to the amount of resources required, both in terms of computing labour as well as memory footprint. Therefore it is desirable to implement specialized kernels for the algorithm at hand to minimize overhead and improve performance. This is especially applicable to neural network applications, where the number of unique kinds of operations is small, such optimization can greatly benefit the overall performance. Some optimized kernels for neural networks are presented in Fig. 9 [13].

In this particular example, the original model is optimized by replacing the convolution, bias, and linear rectifier kernels with a single kernel called Convolution-Bias-Rectifier (CBR), resulting in lower overhead in invoking the kernel. This process is known as vertical fusion. The model is then further optimized by layer aggregation (also known as horizontal fusion), which replaces several same-sized CBR kernels from one input tensor to a single CBR kernel to act on the input tensor. Such optimization reduces the total number of operations by not only reducing the kernel overhead, but also by exploiting the parallelism in same sized convolution layers.



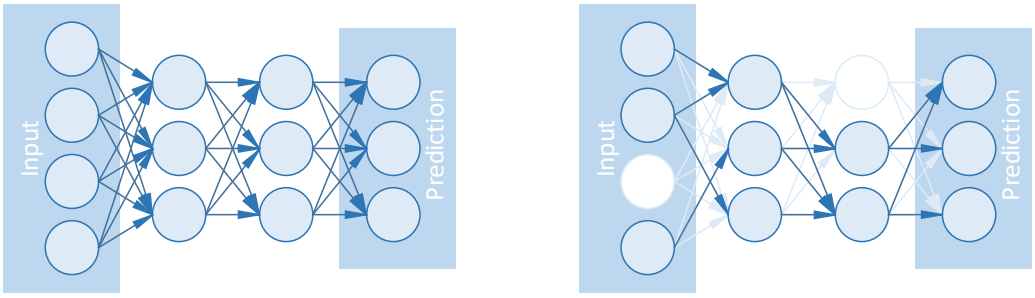
(a) (b) (c)

Fig. 9 - Example of series of neural network optimizations. (a) original model (b) model after vertically fusing kernels and (c) model after layer aggregation.

3.2 *Model Weight Pruning*

Weight pruning (see Fig. 10) refers to the process of removing nodes in the neural network that, relative to other nodes, do not contribute much to the overall outcome of the algorithm. Typically, weights from all layers are gathered and sorted by value, then the weights in the lowest percentile are set to zero. The algorithm then goes through a training phase again to fine tune the remaining non-zero values to maximize accuracy. This process is iterated until a desired threshold in sparsity of the weights is achieved. Sparse weight vectors are typically desired, because if the vectors (or matrices in algebraic terms) are known to be sparse, then optimized sparse matrix

operation kernels can be chosen rather than generic kernels. This is because sparse matrix operation algorithms are typically much faster than their generic counterparts.



(a) (b)

Fig. 10 - Example of network pruning on fully connected layers. (a) original model (b) pruned model with sparse weights

However, in most scenarios the accuracy of the pruned model is lower than that of the original model, and decreasing further as more pruning iterations take place, highlighting an inherent trade-off between accuracy and performance. In this study, weight pruning is not investigated as the models explored are likely too small for weight pruning to have a significant positive effect.

3.3 Precision Calibration

Most operations in neural networks involve floating-point arithmetic. While the precision of floating-point numbers is device dependent, most GPUs in the market today utilize 32-bit floating point precision to represent numbers, which represents numbers in up to 7 or 8 decimal point precision in decimal notation (see Fig. 11).
Another approach to improving the performance of the model prediction is to use lower precision representation of numbers. Because of the highly parallel nature of GPUs, memory I/O cycle at the internal cache buffers is typically high. Consequently, reducing the precision results in reduced memory footprint of the model, and thus requires less memory I/O to make a prediction, and often times leads to faster inference speeds at the cost of lowered precision. In addition, working with lower number of bits typically allows us to increase the clock rate of the processor to further increase the throughput.

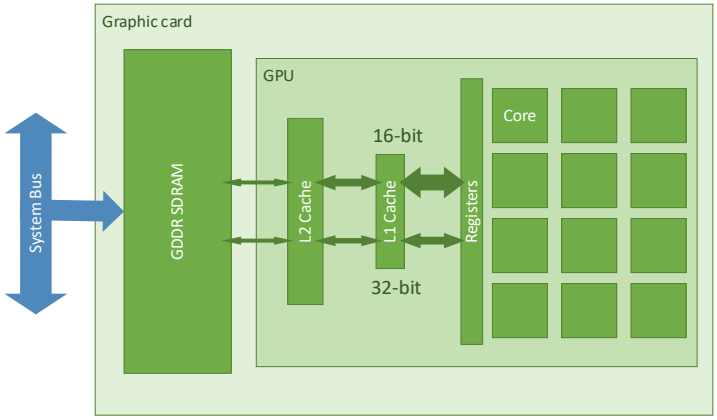


Fig. 11 - Simplified GPU architecture. The capacity to hold and transfer data is doubled in 16-bit mode compared to 32-bit mode

Precision can be arbitrarily lowered, and sometimes can be as low as ternary or binary representation. However, the performance of lowered precision models will depend greatly on the hardware and how it deals with lower precision arithmetic.

3.4 Hardware Accelerators

Hardware acceleration refers to the use of specialized hardware to perform specific tasks. Often the accelerators are comprised of digital circuits that perform a very narrow range of tasks as efficiently as possible, and can be found in many applications such as audio processing systems in mobile System-on-Chips (SoC), image processing systems, optical sensor modules, and encryption systems in network modules. GPUs can also be classified as accelerators for graphics and computing applications. Aside from GPUs, there is a large collection of accelerators for deep learning applications in market today, as shown in Fig. 12.

(a) Nvidia Jetson AGX Xavier SoC | (b) Qualcomm Snapdragon SoC | (c) Intel Movidius Neural Compute Stick | (d) Google Coral

Fig. 12 - SoCs and external accelerators for deep learning and general computing

4. HARDWARE AND TOOLS

As for the platform, we have chosen to use Nvidia Jetson AGX Xavier (see Fig. 13) for onboard processing. The reasons for this selection include:

1. The Xavier is one of a few platforms that allows the configuration of the power output, allowing for benchmarking against different power levels
2. It houses both a GPU and other accelerators specific to neural networks
3. It provides optimized kernels for lower precision operation

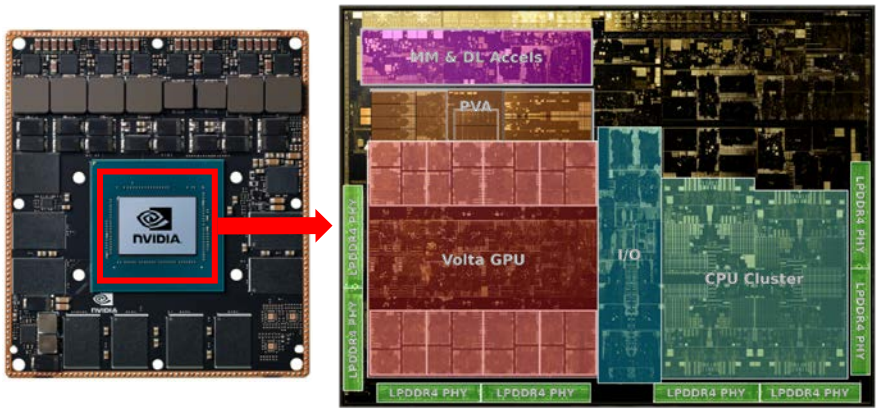


Fig. 13 - Nvidia Jetson AGX Xavier Module and SoC [ADD REF]

Some of the hardware specifications as well as several specialized accelerators for the Xavier SoC are presented in Table 5:

Table 5 - Hardware Specification of Nvidia Jetson AGX Xavier

HW Feature	Jetson AGX Xavier
CPU	8-Core Nvidia Carmel @ 2265 MHz
GPU	512-Core Nvidia Volta @ Max 1377 MHz
GDDR SDRAM	16 GB unified shared memory
GPU L1 Cache	1 MB
GPU L2 Cache	512 KB
Accelerators	2x NVDLA
	2x PVA
	64 Tensor Cores
	Multimedia
Maximum Compute Power	6.3 W @ 10 W setting 8.6 W @ 15 W setting 12.3 W @ 30 W setting

The RAM memory module hosted on Xavier SoC is a unified shared-memory module – the same memory buffers are accessible by both CPU and GPU without a need for data transfer through a serial bus. This minimizes memory I/O between CPU and GPU.

Additionally, the Xavier SoC provides the capability to set the system power configuration to one of three possible values – 10, 15 and 30 Watt configurations. The maximum power that can be brought to computing modules is a fraction of these values – 6.3 Watts for 10 Watt configuration, 8.6 Watts for 15 Watt configuration, and 12.3 Watts for 30 Watt configuration. These values were measured while extensively exercising the GPU onboard the Xavier SoC.

The devices can host several tools provided by Nvidia for optimizing neural networks performance. These tools are listed in Table 6 with explanation of how they can improve the model performance.

Table 6 - List of accelerators and tools employed for optimizing neural networks

Stack	Tool	Description
Machine Learning Framework	TensorRT	Nvidia proprietary framework for designing neural networks. Provides interfaces to optimize neural networks, and automatically applies compute kernel optimization on model at runtime.
Compute Kernels	CUDA	Compute Unified Device Architecture. Nvidia Compute API for programming custom compute kernels.
	cuBLAS	CUDA Basic Linear Algebra Subroutine. Set of optimized kernels for carrying out basic linear algebra calculations such as matrix operations.
	cuSPARSE	CUDA sparse matrix library. Set of optimized kernels for carrying out sparse matrix operations. Used to accelerate calculations involving sparse matrices.

Stack	Tool	Description
	cuDNN	CUDA Deep Neural Networks library Set of optimized kernels for carrying out higher level deep learning calculations like convolution, normalization, and pooling.
Accelerators	Tensor Core	Compute cores specifically designed for optimized, vectorized computations.
	DLA	Deep Learning Accelerator. A module specifically designed for most common neural network operations.
	PVA	Programmable Vision Accelerator. A module specifically designed for computer vision operations such as corner detection and optical flow estimation.
	GPU	Graphics Processing Unit. The Xavier hosts GPU cores with support for optimized reduced-precision operation.

5. ONBOARD EXPERIMENTS AND RESULTS

This section describes a series of experiments with the FCN-1, CNN-1 and CNN-2 models implemented on Xavier SoC device to test the feasibility and effectiveness of performing classification tasks on a low power GPU device. We first deploy our neural network models onto the board under 15W and 30W power settings, and measure onboard performance with metrics introduced in Section 5.1. We then apply compute kernel optimization, precision calibration, and hardware acceleration to the three models on Xavier SoC to investigate the effectiveness of the onboard optimization mechanisms.

5.1 Metrics

Two distinct metrics are selected to profile the power consumption of the hardware used for the experiments. First metric is the system power, which is an estimate on the power required to run the full system hosting the computing solutions. For the Xavier, these values are provided by the vendor, and we present experimental results running under 15W and 30W as the full power supply that Xavier SoC device supplies.

To reflect a more accurate depiction of power consumption required to run the three models, we also make power measurements on target GPUs to benchmark power consumption of the neural network only, since all computing related to the model runs on these target devices. In the case of the Xavier SoC, power measurements can be made at individual component level spanning memory, GPU, and other accelerators including Programmable Vision Accelerator (PVA) and Deep Learning Accelerator (DLA) hosted inside the SoC device.

Aside from power consumption, there are two main metrics used for benchmarking the performance of the model. The first is the throughput, measured in images processed per second. Throughput measures the overall performance of a particular hardware without considering the amount of power being used.

$$Throughput = \frac{Number\ of\ images\ processed}{Time\ took\ to\ process}$$

The second metric is compute efficiency, measured in images per unit of energy. In the literal sense, efficiency measures the rate of computations that can be delivered by the device for every unit of power consumed, or conversely, amount of computation that can be carried out for every unit of energy consumed. Typically, efficiency is obtained by dividing throughput by the amount of power provided. For this study, we define the compute efficiency to be the measured peak compute power divided by the measured throughput. Compute efficiency can best highlight the efficiency of our three models against the computing hardware only.

$$Efficiency_{Compute} = \frac{Throughput}{Measured\ Peak\ Compute\ Power}$$

It should be noted that there are other metrics to benchmark neural network performance, such as TOPS (total number of operations in the network). However, this report focuses on power consumptions for neural networks, and therefore other metrics are not considered for this experiment.

Classification performance (i.e., accuracy and *F1-Score*) is also not considered since the optimization techniques studied here do not affect the classification results. Even for precision calibration optimization, where rounding errors can accumulate throughout the network, we observe that the differences are too small to affect the predicted category. Table 7 summarizes the metrics benchmarked for this study.

Table 7 - Hardware Specification of Nvidia Jetson AGX Xavier

Metric	Units	Description
Throughput	$\frac{images}{second}$	Overall performance of the neural network at a particular configuration. The value signifies the amount images that can be processed in a second.
Compute Power Consumption	*Watts*	Amount of power drawn on computing hardware only, usually a function of time. In this experiment, the peak compute power consumption is reported.
Compute Efficiency	$\frac{images}{second\ Watt}$	Efficiency of the computing components (hardware and neural network algorithm) against compute power consumption. This value signifies theoretical power requirement to run the algorithm. The value is obtained by dividing measured throughput by measured peak compute power consumption.

5.2 Experiment Setup

The models are implemented in Keras (with a TensorFlow backend), an open source Neural Network library. Nvidia provides the TensorRT tool, which is a proprietary framework for implementing neural networks. TensorRT provides interfaces to optimize neural networks, and automatically applies compute kernel optimization on model at runtime. For our onboard optimization experiments, we explore compute kernel optimization, precision calibration and hardware acceleration techniques as described in Section 3. All of these three experiments rely on the TensorRT tool to perform the optimization.

In order to use TensorRT on Xavier SoC device, we serialize each Keras model into Universal File Format (UFF), one of the TensorRT compliance formats for further optimization. There is no direct way to save a Keras model as a UFF file, so we follow the TensorRT guidelines to convert the model to a TensorFlow model, and subsequently save the converted TensorFlow graph to a UFF file [14].

Prior to any onboard optimization experiment, we first convert our Keras models to TensorFlow graphs and perform our baseline benchmarks. The graphs are then serialized to UFF format files for TensorRT to optimize. For each UFF file, TensorRT generates computing engines, which are essentially computing graphs with generic kernels replaced with computing kernels highly optimized for deep learning operations. We again perform benchmarks for the converted TensorRT engines running on Xavier SoC under 15W and 30W as kernel optimized inferences. This is to highlight the contrast between the performances of non-optimized models and optimized models on Xavier.

We then utilize the TensorRT interface to apply further optimizations including precision calibration and hardware acceleration techniques on those engines. The experiments are comprised of measuring each of the aforementioned metrics before and after each onboard optimization technique. It should be noted that, while in theory the optimization techniques are independent, due to the nature of the TensorRT interface, optimizations are applied one after another, and thus it is expected that with each optimization the performance is improved.

Throughput is measured by computing the inverse of average runtime of the model over 50 images that consist of 10 images from each class of objects. The compute power consumption is measured by utilizing Original Equipment Manufacturer (OEM) tool Nvidia Tegrastats [15] for the Xavier device, and taking the peak value during the interval it took to process all 50 images. The compute efficiency is measured by dividing measured throughput by measured peak compute power consumption.

5.3 Results

The benchmarks for all models running on Xavier SoC at different power configurations are presented in Table 8 and Table 9. The throughput and efficiency generally decrease as the algorithms get larger. This is expected, since the number of operations to reach a prediction increases as the number of parameters in a neural network increases, slowing down the inference speed and ultimately lowering the efficiency.

The power consumption measurements on computing hardware (which includes GPU, DLA and RAM memory modules) show a small power usage, from the observation that approximately 1W of power is consumed by GPU and RAM memory collectively while idle. Because the collective compute power consumption never exceeds 2W for these benchmarks, we conclude that the models assessed under this study are too small to utilize the Xavier SoC to a meaningful extent.

Table 8 - Performance Benchmarks Measured on Xavier SoC @ 30W Power Configuration

Model	**FCN**				**CNN-1**				**CNN-2**			
Optimization	**B**	**K**	**PC**	**HA**	**B**	**K**	**PC**	**HA**	**B**	**K**	**PC**	**HA**
Throughput (images/sec)	672	7833	9500	1829	242	5856	6436	1391	217	5604	6785	1259
Peak Power Consumption (Watts)	0.928	1.854	1.854	1.856	1.083	1.701	1.707	1.856	1.238	1.546	1.855	1.856
Compute Efficiency (images/sec/Watt)	724	4225	5124	985	223	3443	3784	750	176	3625	3658	678
Max. Throughput Increase (%)	1317 % with K+PC				2560 % with K+PC				3027 % with K+PC			
Max. Efficiency Increase (%)	607 % with K+PC				1597 % with K+PC				1978 % with K+PC			
B = Baseline, K = Kernel Optimization, PC = Precision Calibration, HA = Hardware Acceleration												

Table 9 - Performance Benchmark Measurements on Xavier SoC @ 15W Power Configuration

Model	**FCN**				**CNN-1**				**CNN-2**			
Optimization	**B**	**K**	**PC**	**HA**	**B**	**K**	**PC**	**HA**	**B**	**K**	**PC**	**HA**
Throughput (images/sec)	285	7045	7817	1400	170	3422	3490	726	163	2860	3171	763
Peak Power Consumption (Watts)	0.928	1.547	1.547	1.392	0.928	1.547	1.547	1.392	0.928	1.547	1.547	1.392
Compute Efficiency (images/sec/Watt)	307	4554	5053	1006	183	2212	2256	522	176	1849	2050	548
Max. Throughput Increase (%)	2643 % with Kernel+PC				2129 % with Kernel+PC				1845 % with Kernel+PC			
Max. Efficiency Increase (%)	1546 % with Kernel+PC				1133 % with Kernel+PC				1065 % with Kernel+PC			
B = Baseline, K = Kernel Optimization, PC = Precision Calibration, HA = Hardware Acceleration												

As the measured power consumption indicates, there is likely much greater room for improvement on throughput and efficiency by invoking the algorithms with batches or streams of input images rather than providing only one input at a time. However, to clearly depict the effects of onboard optimization, it was decided that only one image will be provided at a time for inference for this study. We now present some detailed discussions into the effects of each optimization technique.

5.3.1 Baseline Benchmarks

From Table 8 and Table 9, the baseline throughput and efficiency are by far the lowest for all three models and power configurations. This is due to the fact that TensorFlow (and other generic machine learning frameworks) uses highly generic kernels provided by hardware OEMs to allow for portability across variety of hardware. In contrast, TensorRT kernels are highly specialized for the hardware hosting it, and thus are not portable across devices. However, this allows for optimal utilization of the hardware, as can be seen by non-baseline benchmarks.

As noted earlier and presented below, the larger the model, the less computationally efficient it will be on the Xavier SoC device and likely on other devices. In the following sections, we apply three onboard optimization techniques to see if they can improve computation efficiency of our models on Xavier.

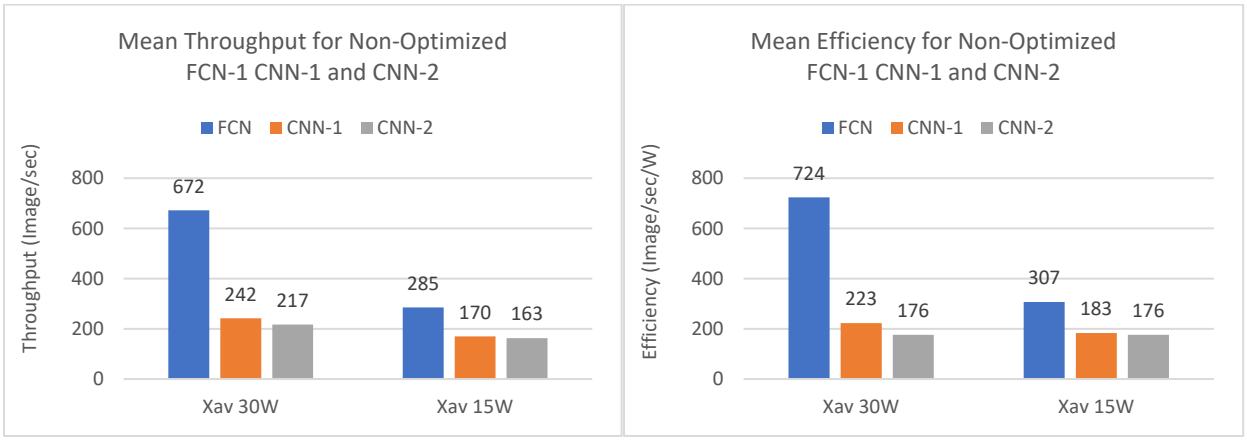


Fig. 14 - Performance benchmarks on Xavier SoC running FCN-1, CNN-1 and CNN-2 models with no optimization

5.3.2 *Compute Kernel Optimization Benchmarks*

Compute Kernel Optimization with TensorRT shows the most improvement in both throughput and efficiency, as demonstrated in Fig. 15. This is expected because Xavier SoC hosts many optimized kernels specifically for deep learning algorithms. Note that not all devices will demonstrate such improvement, as it is not common for OEMs to provide specialized kernels for specific hardware.

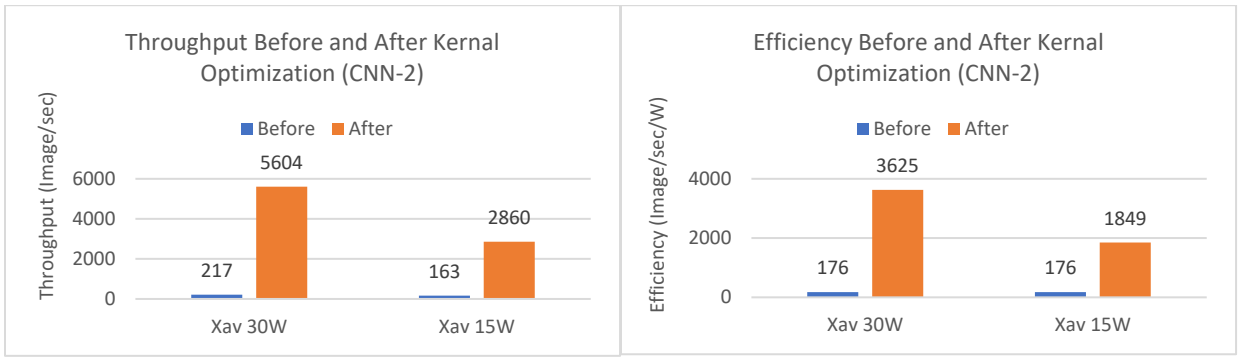


Fig. 15 - Performance benchmarks on Xavier SoC running CNN-2 before and after compute kernel optimization

Also note that this comparison between baseline and kernel-optimized benchmarks may not be an accurate comparison, as they are made from different frameworks (TensorFlow and TensorRT, respectively). Because their interfaces are quite different, it cannot be guaranteed that only the time interval for prediction computation is measured for either cases.

5.3.3 Precision Calibration Benchmarks

In this section, we present performance benchmarks before and after precision calibration on Xavier on all three models. We present the CNN-2 benchmark from Table 8 and Table 9 in a graphical form in Fig. 16 below.

Fig. 16 - Performance benchmarks on Xavier SoC running CNN-2 before and after precision calibration.

From the kernel optimized model, we run the benchmarks before and after applying 16-bit precision calibration onto all layers involving floating point calculations. In general, precision calibration yields only a marginal increase in throughput and efficiency for all three models. This is expected since precision calibration reduces memory footprint and I/O required to run the model, which provides greater benefit to larger models that require a lot of memory I/O. For small models assessed in this study, the benchmarks show that precision calibration is not as beneficial.

However, precision calibration yields some improvements in throughput and efficiency. While not all hardware can improve performance this way, or even employ such techniques, Xavier's compute architecture is better suited for 16-bit operations rather than 32-bit, and this is evident in our observed benchmarks presented above. By computing in 16-bit precision for as many layers as possible in a neural network on Xavier, it is shown that the bigger the model, the more layers qualify for precision calibration, and the better performance improvement the model will achieve.

5.3.4 Hardware Acceleration Benchmarks

For this experiment, we use the 16-bit calibrated model to benchmark performance before and after using onboard accelerators on Xavier. The 16-bit model had to be used in this case because the DLA and PVA accelerators only operate in the 16-bit mode. Additionally, not all operations in our models are supported by these accelerator modules. Specifically, DLA does not support the Softmax operation in FCN-1; DLA does not support the Reshape and the Softmax operation in CNN-1 and CNN-2. Consequently, these unsupported operations have to be offloaded to Xavier's GPU module rather than the non-GPU accelerators, and therefore it is difficult to profile performance for the ideal case in which the entire model is running under accelerators only. We find that majority of calculations are carried out by these accelerators. Fig. 17 shows the performance benchmarks of this experiment, which contrary to expectations show reduced performance with the accelerators. This suggests the portion offloaded to the GPU module may be impacting the benefits of accelerator use.

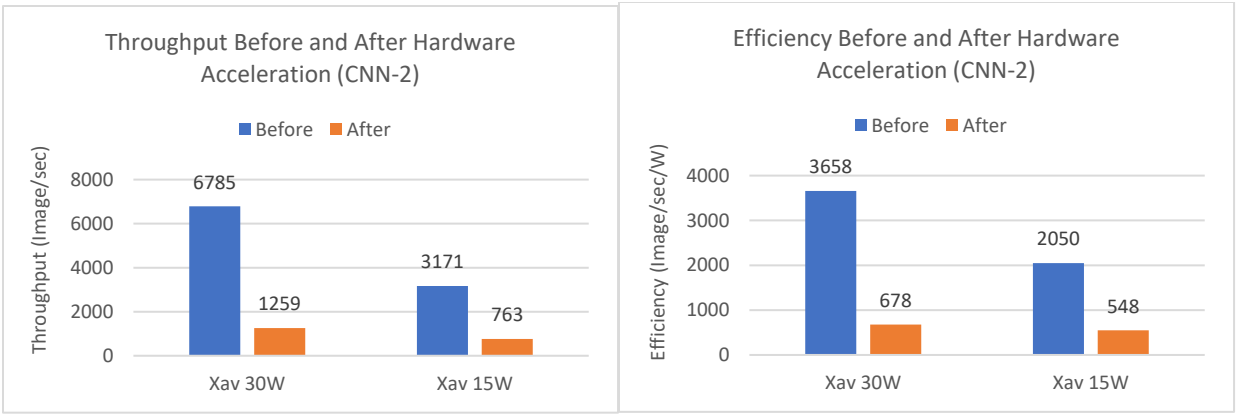


Fig. 17 - Performance benchmarks on Xavier SoC running CNN-2 before and after utilizing Xavier SoC's hardware accelerators.

This optimization mechanism does not provide a benefit in terms of onboard processing if a neural network contains a portion not supported by the DLA and has to be offloaded to the GPU, and the offload procedure may add overhead to the entire computation and downgrade the overall performance. To further investigate the cause, further power consumption profiling takes place over each computing module, as presented in Fig. 18.

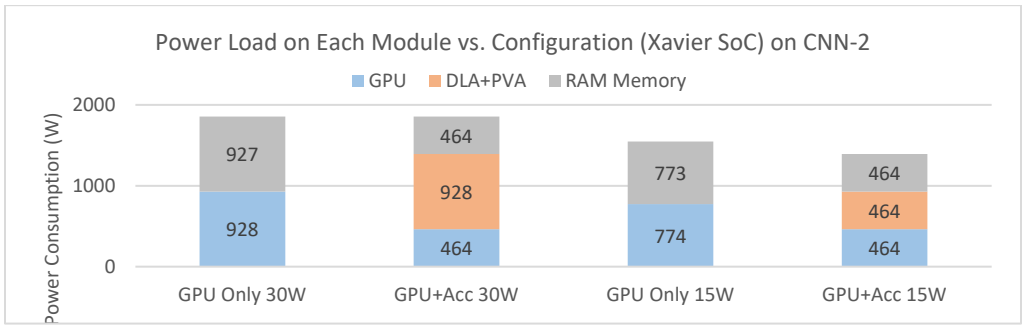


Fig. 18 - Power consumption breakdown on hardware modules in Xavier for CNN-2 model.

There is no significant change in overall power consumption with the introduction of the accelerators. The reduced power consumption on the GPU and RAM memory is offset by the added consumption by the accelerators. This may indicate that memory I/O is moved outside of the memory module presented here, and offloaded to a separate memory buffer used by the DLA, and that the memory buffer for the DLA operates at a slower rate than the memory module used by GPU. However, without further investigation, this remains a speculation.

6. CONCLUSION

To progress towards operationalizing SSA real-time applications, we conducted experiments with modern ML algorithms and technologies for deployment in low SWaP environments. We introduced three simple DNN models, namely, a Fully Connected Network and two Convolutional Neural Networks with different number of hidden layers. These models were used to classify 5 different classes of space objects in MDA's internal Sapphire dataset. The models were first trained and evaluated (with different added synthetic noise levels) using a (off-board) CPU. The trained models were then transferred onboard and various optimization techniques (for onboard inference) were explored.

We observed that the FCN provided faster throughput (compared to the CNN models) due to its simpler architecture and the smaller number of parameters. The advantage of using the more appropriate (for images) CNN models only became apparent as we assessed the models in very low-SNR scenarios. We expect larger (i.e., with larger number of hidden layers) CNN classifiers to be more robust in the presence of noise resulting in improved performance that is more and more significant as the noise levels rise. We believe that this will be an important consideration in developing models in a more operation-ready context. As space surveillance satellites take on a more scalable but

lower-profile configuration, such as that of SmallSats, we anticipate that these platforms will produce images at lower quality compared to the Sapphire platform and thus require more sophisticated algorithms to be operational.

Several hardware platforms and specialized accelerators were explored, and the Nvidia Jetson AGX Xavier was chosen to perform our benchmarks. We introduced several neural network optimization techniques; and applied some of these techniques - compute kernel optimization, precision calibration and hardware acceleration - to our FCN and CNN models. Our benchmarks demonstrate that these optimizations were able to significantly improve the overall throughput and efficiency, with the exception of utilizing the deep learning accelerator.

We observed that the increase in efficiency from these optimization techniques was proportionately greater for larger models under 30W configuration on Xavier. However, the benchmarks in 15W configuration show the opposite trend. Consequently, we cannot form a definitive conclusion about the benefits of optimization as the model gets larger from this study. This is likely due to how small the models are, but further investigation with much larger models is required. In theory, there is a greater potential for parallelization in larger models, resulting in higher throughput rates. This is a key concept to consider, as more sophisticated classifiers and other algorithms related to SSA may greatly benefit from aforementioned optimization methods, beyond the extent presented in this study. This provides an optimistic outlook on the benefits of these optimization techniques for an end-to-end solution with several larger algorithms.

7. FUTURE WORK

We plan to further investigate optimization techniques for SSA applications, particularly in terms of computing efficiency of onboard processing platforms that will include profiling the power consumption of the computing hardware running the models. We will collect these measurements across several devices and reference workstation/server-grade machines to assess the feasibility of utilizing embedded devices onboard spacecraft for SSA applications.

Additionally, we plan to assess the benefits of pruning optimizations to further increase the efficiency of deploying neural networks on embedded devices. This is to achieve minimal resource footprint of this application onboard a spacecraft, freeing up resources to be used for other tasks. Another important future work is to assess the benefit of all these optimization techniques to much larger models compared to those featured in this study, as the power consumption measurements indicate that the hardware modules were not extensively exercised. This raises uncertainty in terms of computing efficiency benchmarks we have obtained for this study, and a more thorough study is required.

Lastly, we will investigate the effects of radiation on system performance, particularly on computing hardware, to gain insight on the various defects that may occur not only in the acquired data, but also on the algorithm itself (such as random bit flips on memory buffer hosting the algorithm), and assess methodologies to mitigate these effects.

8. REFERENCES

[1] D. Girimonte and D. Izzo, “Artificial Intelligence for Space Applications,” in *Intelligent Computing Everywhere*, A. J. Schuster, Ed. London: Springer London, 2007, pp. 235–253.

[2] L. Jenner, “NASA Researchers Teach Machines to ‘See,’” *NASA*, Oct. 30, 2018. http://www.nasa.gov/feature/goddard/2018/nasa-researchers-teach-machines-to-see (accessed Aug. 09, 2020).

[3] “Satnews Publishers: Daily Satellite News.” http://www.satnews.com/story.php?number=1654495662 (accessed Aug. 09, 2020).

[4] K. G. Kelly, “Integrating Machine Learning into Space Operations,” in *AMOS CONFERENCE TECHNICAL PAPERS*, 2017, p. 7.

[5] E. J. Wyrwas, “Body of Knowledge for Graphics Processing Units (GPUs),” Lentech Inc., 2018. [Online]. Available: https://nepp.nasa.gov/files/29564/NEPP-BOK-2018-Wyrwas-GPU-TN60884.pdf.

[6] C. Adams, A. Spain, J. Parker, M. Hevert, J. Roach, and D. Cotten, “Towards an Integrated GPU Accelerated SoC as a Flight Computer for Small Satellites,” in *2019 IEEE Aerospace Conference*, Big Sky, MT, USA, Mar. 2019, pp. 1–7, doi: 10.1109/AERO.2019.8741765.

[7] R. Caruana and A. Niculescu-Mizil, “An empirical comparison of supervised learning algorithms,” in *Proceedings of the 23rd international conference on Machine learning*, New York, NY, USA, Jun. 2006, pp. 161–168, doi: 10.1145/1143844.1143865.
[8] S. Canada, “Space Situational Awareness and the Sapphire Satellite,” *gcnws*, Jan. 30, 2014. https://www.canada.ca/en/news/archive/2014/01/space-situational-awareness-sapphire-satellite.html (accessed Aug. 18, 2020).
[9] Y. LeCun, Y. Bengio, and T. B. Laboratories, “Convolutional Networks for Images, Speech, and Time-Series,” in *The handbook of brain theory and neural networks*, 1998, p. 15.
[10] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” *ArXiv14126980 Cs*, Jan. 2017, Accessed: Aug. 18, 2020. [Online]. Available: http://arxiv.org/abs/1412.6980.
[11] L. Bottou, *Online Learning and Stochastic Approximations*. Cambridge University Press, 1999.
[12] “F1 score,” *Wikipedia*. Aug. 18, 2020, Accessed: Aug. 18, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=F1_score&oldid=973702000.
[13] “TensorRT 3: Faster TensorFlow Inference and Volta Support,” *NVIDIA Developer Blog*, Dec. 04, 2017. https://developer.nvidia.com/blog/tensorrt-3-faster-tensorflow-inference/ (accessed Aug. 10, 2020).
[14] “NVIDIA Webinar — TensorFlow to TensorRT on Jetson,” *NVIDIA Developer Forums*, Feb. 23, 2018. https://forums.developer.nvidia.com/t/nvidia-webinar-mdash-tensorflow-to-tensorrt-on-jetson/58491 (accessed Aug. 28, 2020).
[15] “NVIDIA Jetson Linux Developer Guide: Applications and Tools | NVIDIA Docs,” *NVIDIA Developer Guilde*, Jan, 2019. https://docs.nvidia.com/jetson/l4t/#page/Tegra%2520Linux%2520Driver%2520Package%2520Development%2520Guide%2FAppendixTegraStats.html1 (accessed Aug. 28, 2020).